

How to bring compute intensive C++ based apps to Android

Droidcon, 05/09/2014, Berlin

Martin Siggel

German Aerospace Center (DLR)



Knowledge for Tomorrow



Outline

1. Lets get started
 - Motivation – Why native code on Android?
 - Introduction – About the TiGL App for aircraft design at DLR
2. Hacking
 - Preparing Native App Development
 - Compiling and patching 3rd party libraries
 - Using JNI to communicate between Java and C/C++
3. Building
 - Integrate NDK build into Gradle
4. Testing / Running
 - Using the Android Emulator efficiently




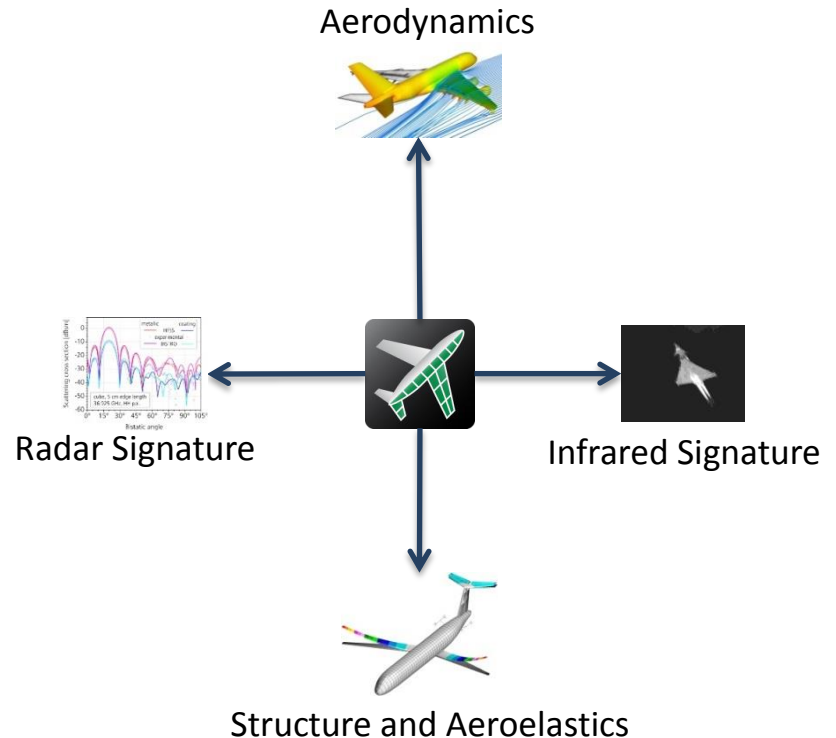
Motivation – Why native code?

- Rewrite of large and tested codebase to Java not reasonable
 - Introduces new errors
 - Code depends on other 3rd party libraries without Java counterpart
 - Too time consuming
- Many numerical algorithms implemented in C or Fortran
- C and Fortran compilers produce highly optimized machine code
- On the other hand:
 - Native code is harder to debug
 - Must be ported (may contain platform specific code)

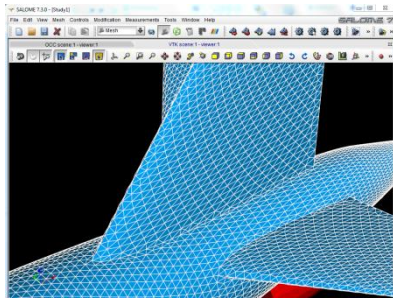


About the TiGL project at DLR

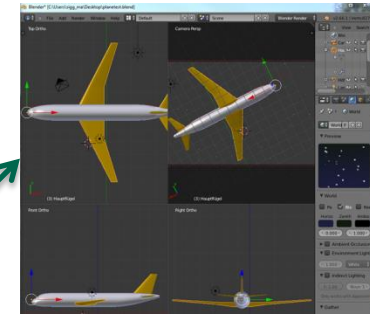
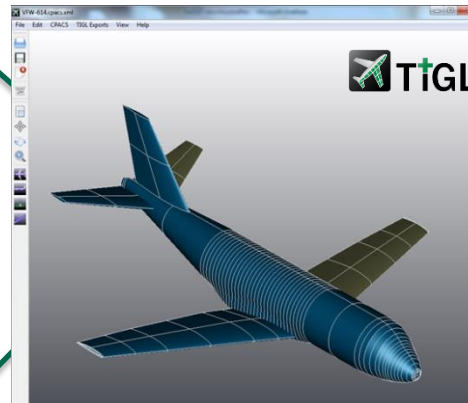
- Central **geometry** viewer and library used by many simulation tools in DLR for **aircraft design**
- Written in C++, large tested code base
- Uses other Open Source Software:
 - CAD kernel OpenCASCADE
 - 3D engine OpenSceneGraph
- Multiplatform – Win, Linux, OS X, and **Android**
- Open Source , Apache 2.0



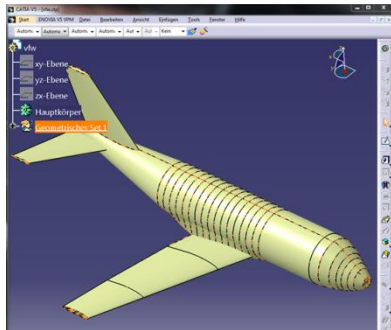
Aircraft design at DLR



Aero/CFD Simulations



Rendering, Visualization



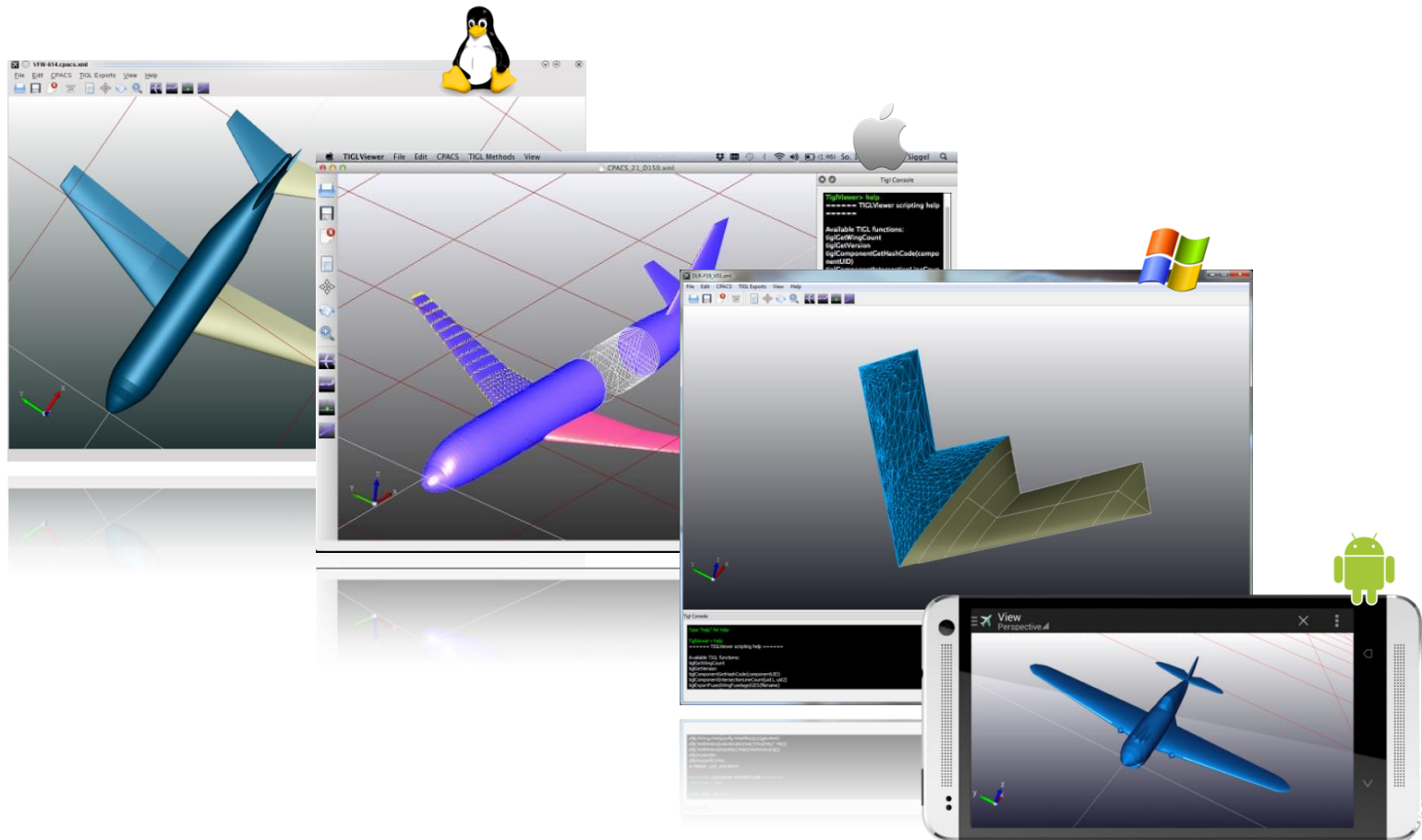
Modeling with CAD Systems



3D Printing



The TiGL Geometry Viewer



Why Android?

Because we can!

And it's fun 😊





What you'll need

- Android SDK
- Android Native Development Kit (NDK):
 - gcc cross compiler toolchains
 - ndk-build: The android build system for native code
 - ndk-gdb to debug native apps
 - Android specific libraries
- Cmake / Autotools to cross-compile dependencies



Steps to the Native Android App

1. Install Android SDK + NDK
2. Cross-Compile and patch 3rd party libraries with NDK toolchain
3. Define a Java Interface class for the communication with your native code
4. Write JNI glue code that calls native code and translates Java objects
5. Create a shared library containing your native code
6. Design Java based Android UI that talks to your JNI wrapper
7. Build the App: Compile native code + standard build steps



Patching OpenCASCADE (and other 3rd parties) code

- Differences in Android's Bionic C library
 - No `timezone` function
 - No `pw_gecos` member in `passwd` struct
 - Missing System V IPC calls, i.e. provide workarounds to
`shmat`, `shmget`, `semctl`,
`semop`, `semget` ...



- Use `#ifdef __ANDROID__` for android specific code paths
 - Deactivate X-Server based code paths
- ⇒ OpenCASCADE rendering engine must be replaced
(it uses X library calls, fixed function OpenGL pipeline ...)



Cross-compiling CMake based 3rd party libs



1. Install standalone cross compiling toolchain

```
$NDK/build/tools/make-standalone-toolchain.sh --platform=android-9  
--install-dir=/opt/
```

2. Create AndroidToolchain.cmake

```
SET(CMAKE_SYSTEM_NAME Linux)  
SET( TOOLCHAIN /opt/arm-linux-androideabi-4.6/)  
  
SET( CMAKE_FIND_ROOT_PATH ${TOOLCHAIN} )  
SET( CMAKE_C_COMPILER      "${TOOLCHAIN}/bin/arm-linux-androideabi-gcc")  
SET( CMAKE_CXX_COMPILER    "${TOOLCHAIN}/bin/arm-linux-androideabi-g++")  
  
SET( CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER )  
SET( CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY )  
SET( CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY )
```

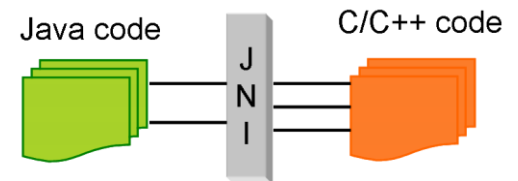
3. Configure CMake project with toolchain argument

```
cmake -DCMAKE_TOOLCHAIN_FILE=AndroidToolchain.cmake ...
```



Use JNI to wrap native code

- **Java Native Interface** enables communication with native (C++) code
- Realized, by calling functions from a shared library (*.dll or *.so)
- Provides:
 - Loading shared library via `System.loadLibrary("mylib")`
 - Functions to convert between Java and C data types
- BUT! - Java to C glue code has to be written by hand
- Header file of JNI Wrapper class should be created with `javah` (assures correct function names, including package names)



JNI – write native interface (Java)

- Define interface class in Java

```
package de.dlr.sc.jnitutorial;

public class NativeInterface {
    static {
        // load tutorial-native.so from libs
        System.loadLibrary("tutorial-native");
    }

    // Declare a native methods
    public native void initNative();
    public native String getName();
    public native void setName(String s);
}
```

- Create JNI header file with javah

```
javah de.dlr.sc.jnitutorial.NativeInterface
```



JNI – automatically generated header file

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class de_dlr_sc_jnitutorial_NativeInterface */

#ifndef _Included_de_dlr_sc_jnitutorial_NativeInterface
#define _Included_de_dlr_sc_jnitutorial_NativeInterface
#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT void JNICALL Java_de_dlr_sc_jnitutorial_NativeInterface_initNative
    (JNIEnv *, jobject);

JNIEXPORT jstring JNICALL Java_de_dlr_sc_jnitutorial_NativeInterface_getName
    (JNIEnv *, jobject);

JNIEXPORT void JNICALL Java_de_dlr_sc_jnitutorial_NativeInterface_setName
    (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```



JNI – handwritten C++ glue code

```
#include "de_dlr_sc_jnitutorial_NativeInterface.h"
#include "tutorial.h"

JNIEXPORT void JNICALL
Java_de_dlr_sc_jnitutorial_NativeInterface_initNative (JNIEnv *, jobject) {
    init_native();
}

JNIEXPORT jstring JNICALL
Java_de_dlr_sc_jnitutorial_NativeInterface_getName (JNIEnv * env, jobject) {
    // create java string from std::string
    return env->NewStringUTF(get_name().c_str());
}

JNIEXPORT void JNICALL
Java_de_dlr_sc_jnitutorial_NativeInterface_setName (JNIEnv * env, jobject, jstring name) {
    // convert to c-string
    const char * cname = env->GetStringUTFChars(name, NULL);
    // call library function
    set_name(cname);
    // free allocated memory for string
    env->ReleaseStringUTFChars(name, cname);
}
```

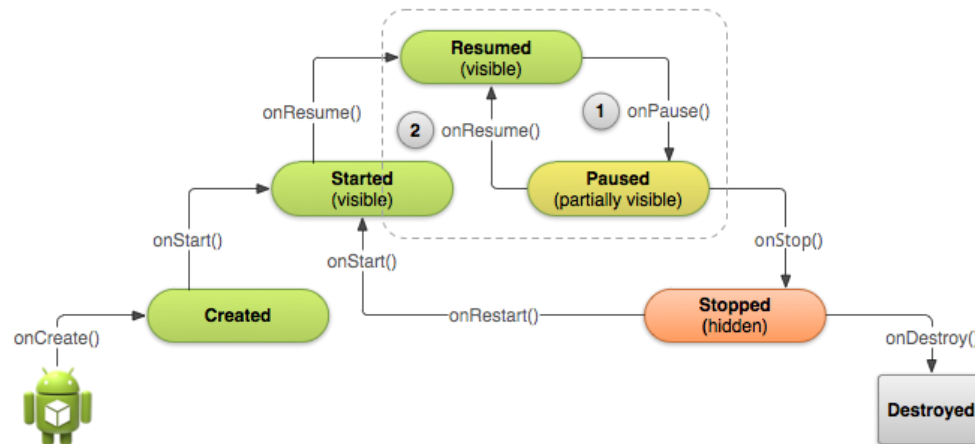


Life cycle of native code

- With hybrid Java/C++ Apps, no automatic life cycle management for native code
- Compared to Java code, no need to store and restore state on suspend/resume

Native library resides in memory

- Provide initialization and shutdown routines for native library



Compiling + Packaging



Automate build with Gradle and NDK

- Gradle?
 - The new build system for Android
 - Replaces Apache Ant
 - Used by Android Studio
- NDK integration into Gradle just recently added by Google
- Problem: Gradle creates Makefiles files which mostly don't work
- Workaround:
 - Disable automatic creation of Android.mk Makefiles
 - Write your own Android.mk files
 - Define a custom build task to call ndk-build



Automate build with Gradle and NDK

1. Register libs folder to the APK and disable automatic Android.mk creation

```
jniLibs.srcDir 'libs'  
jni.srcDirs = []
```

2. Define custom task to build native code

```
task ndkBuild(type: Exec) {  
    def ndkBuild;  
    if (System.properties['os.name'].toLowerCase().contains('windows')) {  
        ndkBuild = new File(System.env.ANDROID_NDK_HOME, 'ndk-build.cmd')  
    } else {  
        ndkBuild = new File(System.env.ANDROID_NDK_HOME, 'ndk-build')  
    }  
    commandLine ndkBuild, '-j', Runtime.runtime.availableProcessors()  
}
```

3. Add task to build dependencies

```
tasks.withType(Compile) {  
    compileTask -> compileTask.dependsOn ndkBuild  
}
```



Running + Testing



Using the emulator efficiently

- Standard Android Virtual Device emulates ARM instructions using QEMU

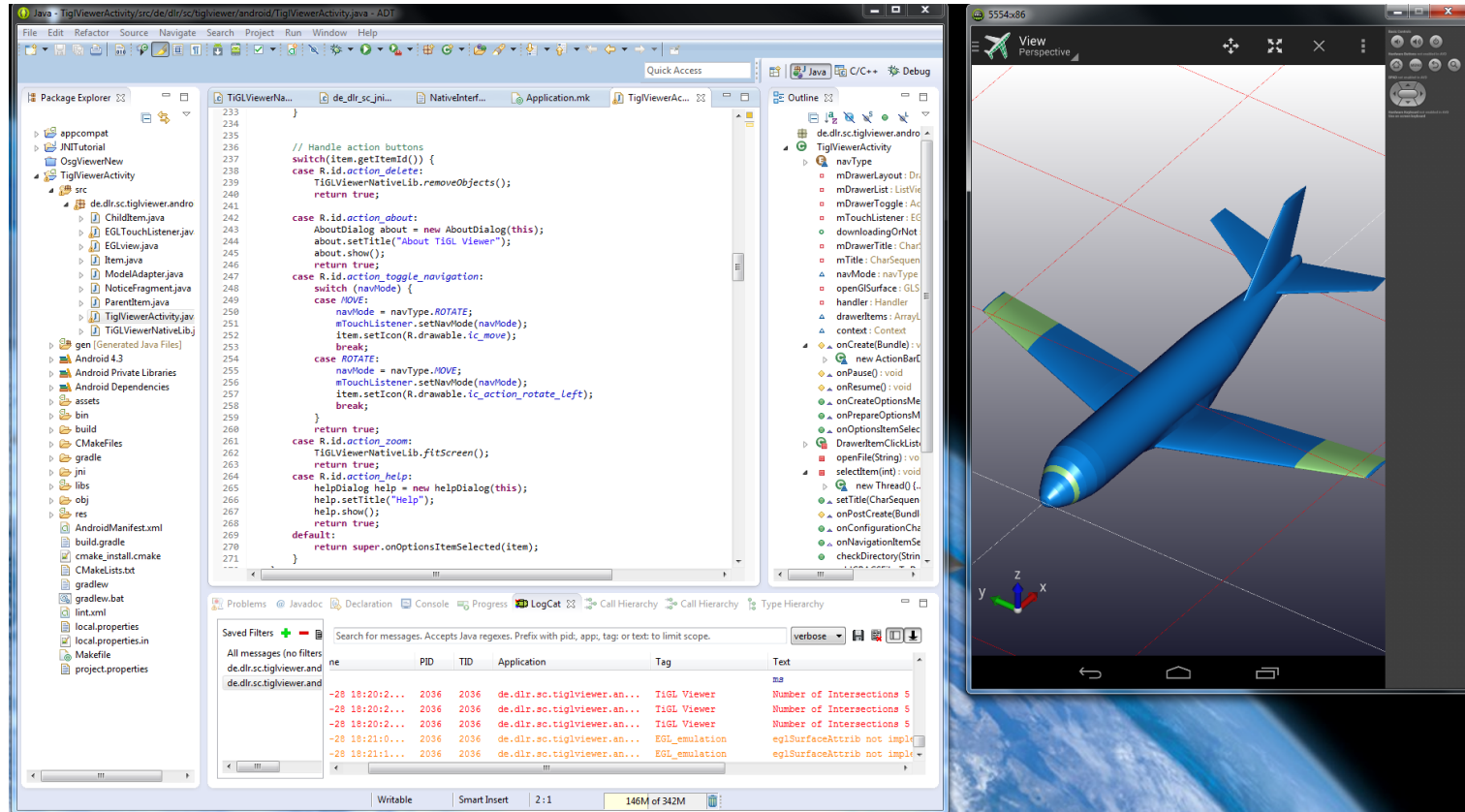


so slow...

- Use a x86 based emulator with GPU acceleration
- Install Intel HAXM acceleration driver (part of Android SDK)
- All 3rd party libraries must be recompiled with an x86 based NDK toolchain
- Add `x86` to `APP_ABI` in `Application.mk` to compile for x86 platform

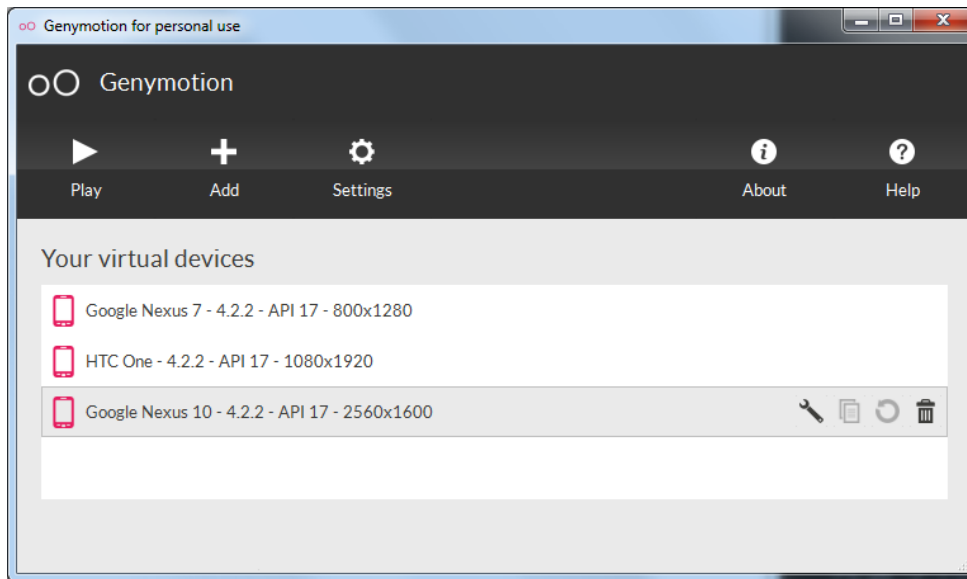


Using the emulator efficiently

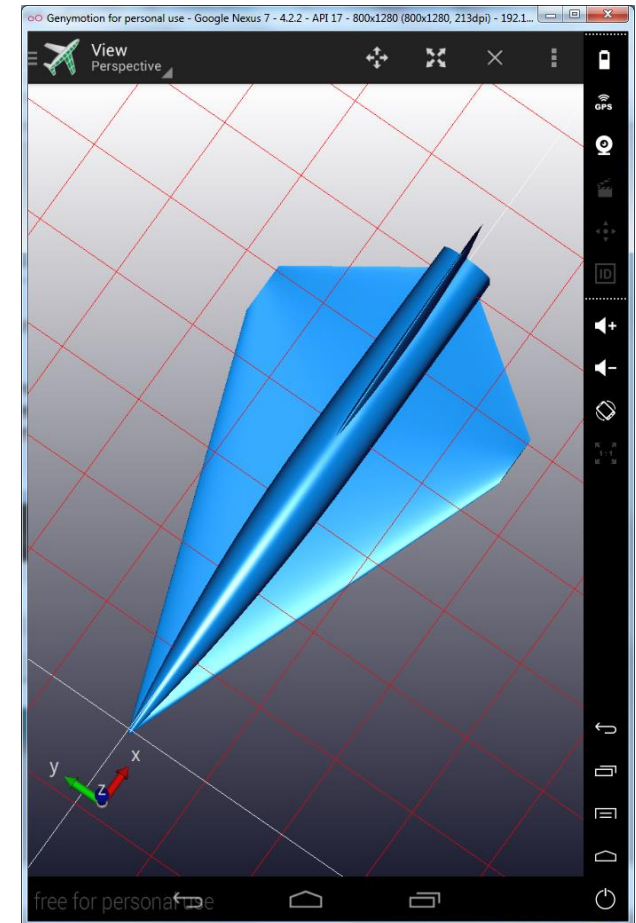


The Genymotion Android emulator

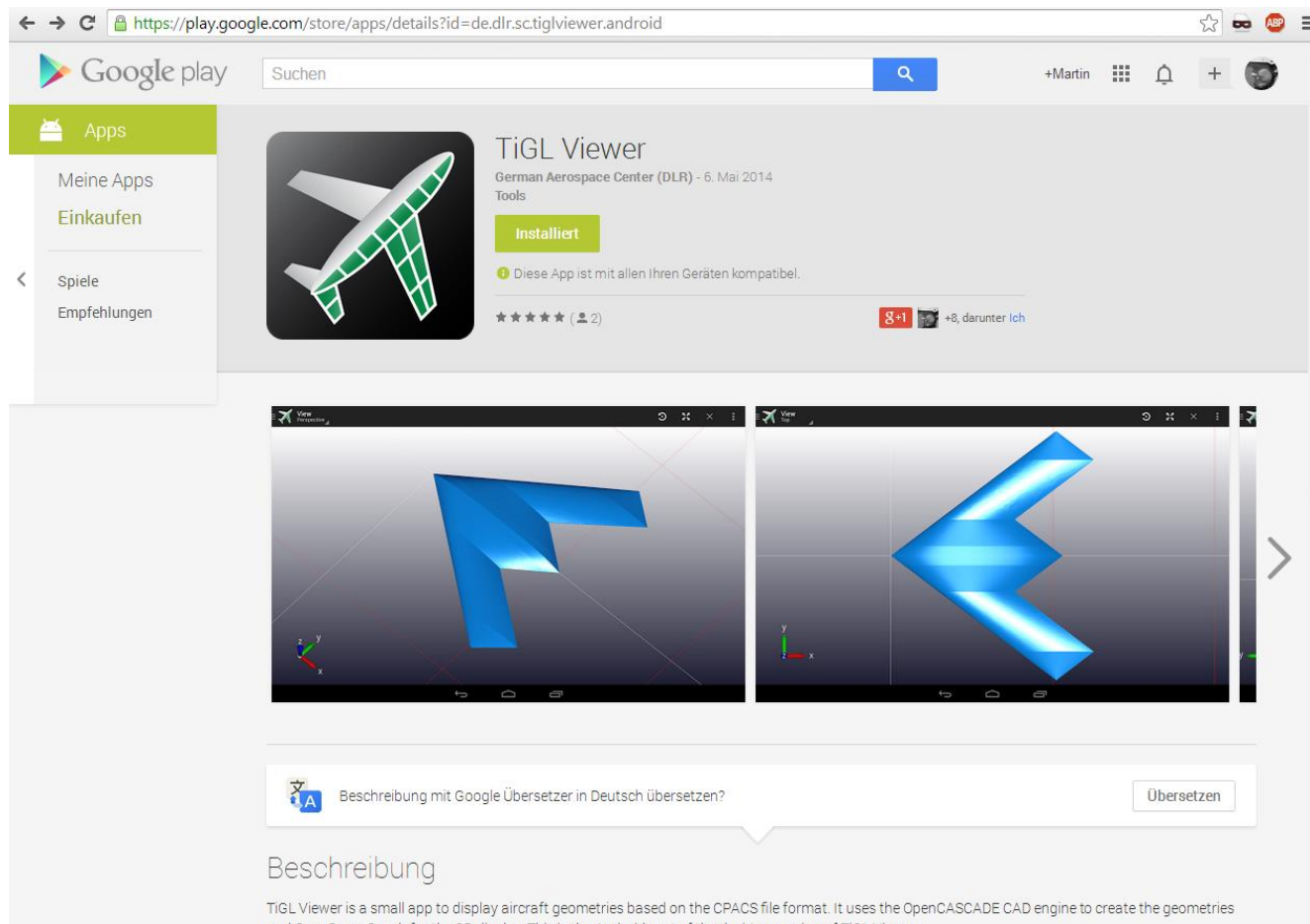
- X86 based very fast Android emulator
- Uses VirtualBox under the hood
- Comes with many pre-configured devices
- Just works



GENYMOTION^{oo}



TiGL Viewer in Play Store



Questions



martin.siggel@dlr.de

https://github.com/rainman110/droidcon_jni



OpenMP Thread parallelism on Android

- Apps crashing when using OpenMP directives outside the main thread
- Reason are differences in Thread Local Storage (TLS) on Android compared to Desktop
- BUT:
 - The NDK can be patched, to enable OpenMP:

<http://recursify.com/blog/2013/08/09/openmp-on-android-tls-workaround>

